Comparing CORBA Implementations

by

Adam Buble

Master Thesis

Faculty of Mathematics and Physics
Charles University, Prague
Czech Republic
1999

Advisor: Petr Tůma
Declaration

This is to certify that I wrote this thesis on my own and that the references include all the sources of information I have exploited.

Adam Buble
August 16, 1999
Acknowledgements

This document has been produced by drawing upon the efforts of a number of people who have contributed ideas. Thanks go to all contributors, but especially to František Plášil, Bernhard Beyl and Stefan Tilkov. Special thanks goes to Petr Tůma.

Iona Technologies has kindly provided an Orbix installation for the purpose of the thesis.

Adam Buble
# Table of Content

Table of Content........................................................................................................4

1. Introduction................................................................................................................6
   1.1. Goal of the Thesis..............................................................................................6
   1.2. Structure of the Thesis .....................................................................................7

2. Overview of CORBA..................................................................................................8
   2.1. Object Management Architecture ......................................................................8
   2.2. Object Request Broker .....................................................................................9

3. Discussion on Comparing .........................................................................................13
   3.1. General Approach to Comparing .....................................................................13
      3.1.1. Analysis of Implementation ......................................................................13
      3.1.2. Benchmarking ..........................................................................................15
      3.1.3. Consequences for This Work ....................................................................16
   3.2. Diversity of Implementations ............................................................................17

4. Commented Test Sheet .............................................................................................20
   4.1. Standardized Functionality ..............................................................................20
      4.1.1. Interface Definition Language ..................................................................20
      4.1.2. Basic Remote Invocation .........................................................................20
      4.1.3. Dynamic Type Manipulation ....................................................................21
      4.1.4. Repositories ............................................................................................21
      4.1.5. Interoperability .......................................................................................21
   4.2. Proprietary Extensions ......................................................................................21
      4.2.1. Communication Extensions .....................................................................21
      4.2.2. Multi-threading Extensions .......................................................................22
      4.2.3. Management Extensions .........................................................................24

4.3. Benchmarks ..........................................................................................................24
   4.3.1. Measurement Verification .........................................................................24
1. Introduction

In 1990, the Object Management Group (an international consortium now consisting of over 800 companies) has introduced its Object Management Architecture (OMA), a standardized architecture for distributed computing in a heterogeneous environment. In OMA, a pivotal role is played by the Object Request Broker (ORB), specified by the Common Object Request Broker Architecture (CORBA).

Over the years, both the CORBA standard (whose current version is CORBA/IIOP 2.3/1.1) and the implementations of CORBA have evolved considerably. Right now, major CORBA vendors offer a range of C++ and Java ORBs that (while usually still CORBA 2.0 compliant) differ in many significant aspects. For a customer who is not intimate with the many OMG standards, it is becoming harder to assess the practical impact of the differences between various ORB implementations.

In the last year, the OMG has realized the need of comparing (or more precisely – benchmarking) the ORBs. The implementations are becoming so much different (primarily with respect to performance) that there is a strong need to compare implementations mutually. OMG has moved forward to establish Benchmarking Platform Special Interest Group (PSIG) that should standardize ways to benchmark implementations.

1.1. Goal of the Thesis

It is a goal of this thesis to identify a set of criteria that would help to evaluate an ORB implementation, and to evaluate a set of ORBs using these criteria. The criteria should cover those aspects of the ORB functionality that are important from the application developer’s point of view (this includes functionality specified in the OMG standards, together with useful proprietary extensions). For measurement and testing purposes, a set of IDL definitions is to be specified and some sample benchmarks are to be developed.
1.2. **Structure of the Thesis**

Chapter 2 introduces reader into the world of OMG standards, particularly into the CORBA specification. The very basic concepts of CORBA architecture are outlined.

Chapter 3 shows the simplified structure of a CORBA implementation that will be used for evaluation. The problems related to benchmarking and important consequence for the thesis are stated here. At the end of chapter, I show possible variation of implementations – the features that should be evaluated to obtain good assessment.

Chapter 4 presents the core of the thesis – it forms the test sheet with comments. A user who wants to use the thesis to evaluate an ORB implementation can, based on this chapter, easily appraise the implementation. The criteria presented here are selected based on the analysis in the previous chapter.

Chapter 5 concludes the thesis. Bibliography reference follows. As an appendix to the thesis, a sample evaluation of several CORBA implementations is presented.
2. **Overview of CORBA**

This chapter gives a short introduction to those readers that are not familiar with CORBA structure and functionality. The readers intimate with CORBA may freely skip the chapter.

2.1. **Object Management Architecture**

The Object Management Group, Inc. (OMG), founded in 1989, is an international organization grouping system vendors, software developers, and end-users. The main goal of OMG is to promote object-oriented technology by defining "a living, evolving standard with realized parts, so that application developers can deliver their applications with off-the-shelf components for common facilities like object storage, class structure, peripheral interface, user interface, etc." [1]. To achieve this goal, OMG has defined the Object Management Architecture (OMA) characterized by definition of two models: the Core Object Model (and its extensions called profiles) and the OMA Reference Model.

OMG's understanding of the object-oriented paradigm is reflected in the Core Object Model. The model defines such concepts as object, inheritance, subtyping, operations, signatures, etc. Additional concepts can be added to create an extension (component). A component should not replace, duplicate, and remove concepts. Components should be orthogonal to each other. A profile is a combination of the Core Object Model and one or more components (typical examples of profiles are e.g. the CORBA profile, the ODP Reference Model).

With the intention to provide a broad object-oriented architectural framework for the development of object technology, OMG defined the OMA Reference Model, published in 1992. The OMA Reference Model is comprised of the following five components:

- The Object Request Broker component serves as a means for delivering requests and responses among objects. It is a backbone of the OMA Reference Model, interconnecting all the remaining components.
The Object Services component provides a standardized functionality (defined in the form of object interfaces), e.g. for class and instance management, storage, integrity, security, query, and versioning.

The Common Facilities component defines a collection of end-user facilities (such as a common mail facility) that a group of applications is likely to have in common. In the current version of the OMA Reference Model, however, are the Common Facilities suppressed.

The Domain Interfaces component contains a set of application-domain-specific interfaces (such as a support for healthcare or electronic commerce).

The Application Objects component is not standardized by OMG. Application Objects use Domain Interfaces, Common Facilities, and Object Services via the Object Request Broker.

2.2. Object Request Broker

CORBA was adopted by OMG in December 1991 as a joint submission of DEC, HP, etc. The first version was denoted as CORBA 1.1 and was, in 1994, replaced by CORBA 1.2 which brought more or less "cosmetic changes" with respect to CORBA 1.1. A much more important development step was CORBA 2.0, which was adopted in December 1994 and which overcomes the drawbacks of CORBA 1.2 by specifying ways for interconnecting different ORBs, and also by suggesting means for establishing interoperability between CORBA-based and non-CORBA-based environments, such as Microsoft’s COM/OLE. CORBA 2.1 followed in August 1997, CORBA 2.2 in February 1998 and CORBA 2.3 in December 1998 (which is the current standard) last updated in July 1999.

![Structure of the CORBA ORB](image)

**Fig. 1.** Structure of the CORBA ORB

The structure of a CORBA ORB is depicted on Fig. 1. The OMG standard defines the structure in terms of interfaces and their semantics. In an actual implementation, however, there is no obligation to reflect the
structure; it is only necessary to preserve the interfaces and their semantics.

Interface of CORBA objects is defined using Interface Definition Language (IDL). Such definition is translated using compiler into particular language (the mappings are standardized by the OMG). Compiler generates stubs, skeletons, dispatching information or entries into the Interface Repository (see later).

**Client Side (Stubs, DII, Interface Repository)**

To request a service from an object implementation, the client can use either static or dynamic invocation. The static invocation mechanism (SII) requires the client to know the IDL definition of the requested service at compile time. The definition is used to automatically generate stubs for all requestable operations. To request an operation, the client calls the appropriate stub, which passes the request to the ORB Core for delivery.

As opposed to static invocation, the dynamic invocation mechanism requires the client to know the IDL definition of a requested service at run time. To request an operation, the client specifies the target object, the name of the operation, and the parameters of the operation via a series of calls to the standardized Dynamic Invocation Interface (DII). Again, the request is then passed to the ORB Core for delivery. The semantics of the operation remains the same regardless of the invocation mechanism used. To obtain an IDL interface definition at run time, the client can use the Interface Repository. The repository makes IDL definitions (e.g. module, interface, constant and type definitions) available in form of persistent objects accessible via standardized interfaces. The ORB is responsible for finding the target implementation, preparing it to receive the request, and communicating the data of the request. The client need not care about the location of the object, language of the implementation, or other things not described in the interface.

On the client side, a target object is usually represented as a proxy object [6] supporting the same interface as the target object. In the case of static invocation of an operation \( m \), the proxy’s method \( m \) calls the stub associated with \( m \); the stub is responsible for creating the request and passing it to the ORB Core for delivery to the target object. In case of dynamic invocation, the client’s code creates the request dynamically by calling the DII interface; in principle, no proxy and stub are necessary.

**Implementation Side (Skeletons, DSI, Implementation Repository, Object Adapters)**

Again, either the static or the dynamic mechanism can be used to convey the request to the actual service implementation. The static mechanism (SSI) requires the IDL definition of the requested service to be available at compile time. An automatically generated operation-specific skeleton is then used by the ORB to call the implementation of the requested service.
The dynamic request delivery mechanism expects the implementation to conform to the standardized Dynamic Skeleton Interface (DSI). Through this interface, the ORB dispatches the request to the target object (to the service implementation). In analogy to the dynamic invocation mechanism on the client side, no compile time information about the interfaces is needed. The information necessary to locate and activate implementations of requested services is stored in the Implementation Repository. The Implementation Repository is specific to a particular ORB and environment and is not standardized by CORBA.

To access services provided by the ORB itself, an implementation uses the Object Adapter. To satisfy the needs of diverse environments, the ORB can be equipped with several different object adapters. Each ORB, however, must provide the standardized Object Adapter interface – Basic Object Adapter (BOA) for pre-CORBA 2.2 ORBs and Portable Object Adapter (POA) for CORBA 2.2 ORBs. The operations of the Object Adapter include generation and interpretation of object references, authentication of clients, and activation and deactivation of target objects.

**ORB Interface**

For accessing the general services provided by an ORB, both client and implementation use the ORB interface. The operations of this interface include converting object references to strings and vice-versa, determining the object implementation and interface, duplicating and releasing copies of object references, testing object references for equivalence, and ORB initialization.

**ORB Core, Interoperability**

The initial CORBA standard left the implementation of the ORB Core totally to the ORB vendor. It was hence almost impossible to interconnect two or more CORBA-compliant ORBs released by different vendors. This lack is remedied by the CORBA 2.0 specification that defines two approaches for achieving interoperability: the General Inter-ORB Protocol (GIOP) and Inter-ORB bridging.

The General Inter-ORB Protocol specifies a set of low-level data representations and message formats for communication between ORBs. It is designed to be simple enough to work on top of any underlying connection-oriented transport protocol, such as TCP/IP and IPX/SPX. The GIOP based on TCP/IP is called the Internet Inter-ORB Protocol (IIOP). CORBA 2.0 requires GIOP as the mandatory protocol. To allow "out of the box" interoperability, CORBA 2.0 specifies the optional Environment Specific Inter-ORB Protocol (ESIOP) as an alternative to the GIOP. It has been mainly specified to allow integration of DCE with CORBA (DCE/ESIOP).

The main idea of bridging lies in mapping requests from one vendor’s format to another when crossing ORB boundaries. Bridging can be performed either at the ORB Core level as an ad hoc solution for every ORB-ORB pair (in-line bridging), or at the application level (request-level
bridging). Request-level bridges mediate requests by utilizing DII and DSI mechanisms to dynamically create and dispatch requests. Bridges are currently mainly used for connecting CORBA with non CORBA-compliant platforms, such as COM/OLE.
3. **Discussion on Comparing**

In this chapter, I discuss the possible ways of evaluating CORBA implementations, the problems related to benchmarking, and the parts of a CORBA implementation that can be evaluated. The discussion serves as a groundwork for the evaluation framework presented later on.

Following the discussion, I go over the CORBA implementation structure once more and show the possible variation of implementations (to illustrate what is worth evaluating).

3.1. **General Approach to Comparing**

3.1.1. **Analysis of Implementation**

Nowadays, a CORBA implementation includes not only the functionality standardized by OMG, but also some proprietary extensions. The extensions often provide important functionality and cannot be simply ignored because they are not a part of the CORBA standard. Many of the extensions also persist from earlier version for backward compatibility reasons, and the implementation often provides standardized ways to achieve the same functionality.

All parts of the implementation defined by OMG, and those extensions with clearly defined purpose, can be evaluated with respect to defined vs. provided functionality. In case of proprietary extensions, however, this is not as straightforward as with the standard parts, because the code is not CORBA-compliant anyway. Other very useful and interesting results can be obtained from benchmarking of the implementation.

Based on the functionality and type of measurement the tests can be partitioned as outlined on figure (Fig. 2).
<table>
<thead>
<tr>
<th>functionality</th>
<th>performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>standardized</td>
<td>A</td>
</tr>
<tr>
<td>extensions</td>
<td>B</td>
</tr>
</tbody>
</table>

**Fig. 2.** Partitioning of the implementation tests (into classes)

- Classes of functionality (A and B) are evaluated using *classifying*. Each part of the implementation can be classified by a value from a predefined range.
- Classes of performance (C and D) are evaluated using *benchmarking*. For detailed discussion on benchmarking, see section 3.1.2.

**Class of Standardized Functionality**

Functionality of all the parts of a CORBA implementation defined by OMG belongs to the evaluation class A (standardized functionality). Within the class, it is possible to classify the provided level of functionality (e.g. by an integer value from a predefined range, which makes an easy comparison of implementations possible).

The IDL language can serve as an example here. The compliant syntax and mapping into a programming language can be classified. In addition, limitations of the IDL language or the IDL compiler can be evaluated.

**Class of Functionality of Extensions**

Within the class B (functionality of extensions), only a few parts of a CORBA implementation can be compared – specifically, those parts whose presence can be predicted and is common among more implementations, and that are not standardized yet. Many useful features that used to be proprietary have now became standards.

This part of the ORB functionality is difficult to compare, because of the possible variability in the features of individual implementations. If a comparable part can be specified, however, it can then be classified in a way similar to class A.

The extension belonging to this class can be e.g. the multi-threading extension. The commonly used CORBA standard, version 2.0, does not specify the ways of achieving multi-threading. Therefore, vendors use proprietary mechanisms to provide an ORB that runs multi-threaded. The ways of supporting multi-threading and all the offered models can be evaluated here.
Class of Performance of Standardized Parts

Performance of the standardized parts of an implementation can be evaluated using benchmarks. For more detailed discussion on benchmarking please see section 3.1.2.

The benchmarks usually evaluate several of the ORB’s parts at once. It is generally impossible to separate some of the ORB parts for the measurement purposes without modifying the ORB (e.g. marshalling, dispatching, unmarshalling are always involved). Hence, a benchmark usually evaluates overall performance, although the load imposed on the ORB by the benchmark can target a specific part of the implementation.

Class of Performance of Extensions

This class, even more than then the class B, is very hard to evaluate and compare. The same problems as in class B (and furthermore as in class C) arise here. I will evaluate parts of class D only in combined features of the ORBs.

Combined Features

Combined features of the implementation are those features that utilize more parts together to achieve a more complicated, but usually useful, result. Such a feature can be e.g. the ability to provide load balancing, scalability or robustness. These are not handled by any individual part of the ORB implementation or the object services; several simpler features must be used together to achieve the expected result.

Combined features can be classified or benchmarked as any other ORB feature. Attention should be paid here because a typical combined feature is complicated and relies on a number of factors.

An example of such a feature can be scalability, which is often discussed, in CORBA-related talks. Various scalability types can be benchmarked: scalability with respect to number of objects, resource consumption, etc.

3.1.2. Benchmarking

In computer science terminology, a benchmark usually denotes a test that evaluates performance of the software system. As wider meanings or even other meanings of the words benchmark and benchmarking are used, I define how I will use the two terms:

- **benchmark** – a test application that produces results that can be used to evaluate the test platform (a result is typically a number indicating time, amount, etc).

- **benchmarking** – running benchmarks and evaluating their results (possibly summing them into the chart or table).

There is a number of issues that should be noticed. Here, I will point out the basic, and from my point of view the key, ones. For more information
on this topic, please see [11] and individual responses ([12], [13], [14], [15], [16], [17], [18], [19]) to the Benchmarking PSIG’s Request for Information [10].

The benchmark would ideally evaluate a well-defined part of an implementation, or a specific action. The latter approach is especially useful in case of testing a well-known or important feature. The first approach, however, is also valuable, as from the known performance of smaller parts of the implementation, the overall effect can be deduced. Moreover, in cases where certain part of the implementation will be used more that the others, one can emphasize the result obtained from that part when comparing the ORB implementation.

The idea of measuring of smaller parts of an ORB implementation is outlined in our group’s response to the benchmarking RFI [12].

The measurement of smaller parts can be done in two ways:

- **indirectly** – the effort of more parts together is measured. By focusing the load so that it has the highest impact on the selected part, I can find out the performance of that part.

- **directly** – within the ORB, a special instrumentation that would indicate entering or leaving the observed part must be present.

An output of a benchmark is usually closely tied to the configuration of the system on which the benchmark runs. For example, the time to ping from the client to the server depends on the hardware of machines where both the client and the server applications run, on the type of transport between those machines, and on the ORB configuration. Mutual comparison between results from different configured system is nearly impossible (as our attempts within the CORBA and Distributed Systems Research Group indicate).

The solution should be to select a common metric to which every benchmark result can be converted. Such a metric is dependent on every possible platform where the benchmark could run. The simplest approach is to run all benchmarks on a single platform, as no conversion of results needs to be done. On the other hand, such results do not tell much about the ORB performance on other platforms.

### 3.1.3. Consequences for This Work

The classification presented in section 3.1.1 (i.e. classify using integers for functionality categories) is useful when we want to present a general comparison of the CORBA implementations as a whole. In order to reach a verdict on the comparison, however, we need to know the weights of individual parts with respect to typical usage. These weights generally vary, and to discover them would need huge empirical research across all possible CORBA application domains. I therefore avoid comparing the implementation as a whole and only benchmark each part of the ORB implementation individually. A user can then choose the implementation
that best suites his needs by assigning the weights to the results based on his requirements.

The benchmarks that I propose in this thesis are mostly of the kind that evaluates a complete action. Most of them, however, are designed so that is possible to find out impact of smaller parts of an implementation (e.g. dispatching). Considering the problems with relevant metric, I will benchmark on a single system configuration only.

3.2. Diversity of Implementations

In line with the previous chapter, I will discuss each part of a CORBA implementation and show the differences that can appear in different implementations and be subject to benchmarking or classification. As the subject of the thesis is CORBA, I will not focus on other parts of OMA (namely on Object Services).

Client Side

On the client side, stub, ORB interface (see later), DII and Interface Repository can be evaluated.

The interface of the stubs is standardized by the appropriate language mapping. Most of the language mapping standards have reached a stable form some time ago and are usually observed by the individual ORB implementations to the letter. Thus, I will not classify the stubs, but I will benchmark performance of marshalling the request and unmarshalling the reply.

DII is also standardized completely by OMG, compliance with standard can be classified and benchmarked. As the DII is used by a small number of real-world applications (typical CORBA application uses static invocation interface\textsuperscript{1}), I will not benchmark DII in any detail.

The Interface Repository is mostly used with DII and therefore I will not evaluate it in any detail either. If it were to be evaluated, it could be done both through classifying (it is fully standardized by OMG) and benchmarking (speed issues).

Implementation Side

The object implementation side affects the performance of the CORBA-based application most of all. This is because the Object Adapter must dispatch incoming request, potentially start server application (involves interoperaction with Implementation Repository) and unmarshall the request.

\textsuperscript{1} I state this based on my experience, and on the fact that omniORB became a very popular CORBA implementation with no support for DII/DSI.
The skeleton code on the server side is similar to the stub on client side. The interface of the skeleton is standardized by OMG and usually observed by ORB vendors. Hence, only the performance issues will be benchmarked (unmarshalling/ marshalling/ dispatching).

As far as the evaluation goes, DII is similar to DII in that it is of little importance except for very special applications. A typical application already knows at compile-time the interfaces it will provide. As such, I will not evaluate DSI in any details either.

Unlike the Interface Repository, the Implementation Repository is an important part of the CORBA implementation [7]. The Repository is usually shared within a domain (e.g. single machine, local network). It keeps the information about all (registered) servers – whether the server is running, server is in the persistent state, server needs to be launched manually, or it can checked the rights to launch the server, etc. As such, it is contacted by Object Adapter on every connection attempt and potentially even during client – server communication.

Neither the structure nor the communication protocol of the Implementation Repository are standardized. The Repository mostly affects the binding phase, where the performance issues can be evaluated.

Finally, the Object Adapter dispatches each incoming request to the server. As such, it is crucial to the performance. Its functionality is commonly known and easy to implement. However, there can be differences that can affect the overall performance (especially from scalability point of view).

**ORB Interface**

The ORB interface is well specified by the CORBA standard. Potentially, some performance issues can be evaluated here, but ORB itself cannot be reasonably tested, as its influence is included in every other test. I therefore do not create any benchmark to test ORB only.

**Interoperability**

In CORBA 2.0, the use of IIOP as basic communication protocol is not mandatory, a CORBA implementation can use a proprietary protocol. Without IIOP, however, a CORBA implementation cannot be connected to the rest of the CORBA-world, which makes it rather useless. Most of today's ORBs use IIOP as its native communication protocol. This guaranties mutual interoperability between these implementations. However, the IIOP implementations can differ (e.g. IIOP versions 1.0 and 1.1) and can handle certain situation in an incompatible manner (e.g. establishing connections, exception handling).

**Others**

There are some other aspects that were not discussed yet and are interesting to evaluate. These do not relate directly to any part of a
CORBA implementation. Examples include overall speed (that can be evaluated as throughput, latency or scalability), load-balancing, multi-threaded extensions, etc.
4. **Commented Test Sheet**

Every evaluation should start with basic information about the implementation. To the most interesting belongs:

- ORB name
- Vendor
- Version evaluated, possibly with release date
- Platform used – the computers and network configurations. Other important configuration such a compile options can be shown as well.

I suggest to include one more test sheet into the final comparison, on which mutual comparison of each involved implementation will be shown.

4.1. **Standardized Functionality**

4.1.1. **Interface Definition Language**

This section focuses on the support for the Interface Definition Language within the ORB, including its mapping to the implementation language (C++ in my case). Some of the limitations of the IDL compilers with respect to the structural complexity of the IDL sources are mentioned as well (these are obtained using the IDL.NAMES, IDL.LONG and IDL.DEEP benchmarks, see section 4.3.2).

4.1.2. **Basic Remote Invocation**

This section focuses on the basic remote invocation functionality of the ORB (e.g. support for remote calls, exceptions, contexts, object adapter functionality). The section also includes the following three groups of benchmarks:
4.1.3. Dynamic Type Manipulation

This section lists the ORB’s support for manipulation of dynamically defined types, that is DSI, DII, and for CORBA 2.2 ORBs also the DynAny Interface.

4.1.4. Repositories

This section describes the Interface Repository and the Implementation Repository of the ORB.

4.1.5. Interoperability

The IIOP protocol support is evaluated in this section. I use the THROUGH.IN benchmark (see section 4.3.3) to test the interoperability of different ORBs (the test passes a wide variety of data types between the client and the server). Interoperability with non-CORBA-based environments (e.g. Microsoft COM, Java) is briefly outlined as well.

4.2. Proprietary Extensions

4.2.1. Communication Extensions

Locating Objects

One of the very basic features required by ORB clients is the ability to locate an object given a set of criteria (e.g. server host, interface). In the standardized OMA architecture, this feature is to be provided by the Trading Service, but since this service was standardized relatively late (August 1996, which is 5 years after the first CORBA standard appeared), many vendors have introduced proprietary solutions in the meantime.

Binding to Objects

The standard way to bind to an object is to get a bootstrap reference to the Naming or Trading Service and to use that reference to obtain whatever other references are necessary. Alternatively, the string_to_object() operation of the ORB can be used to import stringified object references. In addition to (or in place of) these functions, the ORB vendors often introduce proprietary binding mechanisms.

Instantiating Implementations

The instantiation of a CORBA object on the server side of an application consists of:

1. creating of the actual implementation object
2. creating of the unique object reference
3. registering of the implementation object with the reference
The mechanism to do this was described poorly in the original CORBA standard (this is fixed in CORBA 2.2) and thus its implementations differ in the individual ORBs. The same goes for the object deletion mechanism.

**Persistent References**

An important feature of the ORB is the support for persistent object references. These are necessary when implementing persistent objects and are usually tied to some sort of a server activation mechanism. Again, the original CORBA standard is not detailed enough in this area (fixed in CORBA 2.2) and thus most ORBs offer proprietary solutions.

**Automatic Activation**

An important part of the ORB functionality, used especially with persistent servers and servers that are to provide a higher degree of reliability, is the ability to launch a server on demand. This feature is mostly implied by the existence of the Implementation Repository, although different ways of achieving automatic activation can be used.

**Automatic Reconnection**

Related to the automatic activation mechanism is a mechanism that allows a client to automatically reconnect to a server in case of a communication failure.

**Invocation Extensions**

Extensions to the standard invocation mechanism (e.g. asynchronous invocations, callbacks, proxy objects modified by programmer) are provided by most ORBs as well.

**Customizability**

It is also important to have the possibility to customize the ORB behavior (e.g. intercepting and filtering of messages, piggybacking of data onto messages, event hooks).

4.2.2. *Multi-threading Extensions*

**Multi-threaded Servers**

The most common case of a multi-threading extension is a support for multi-threaded server implementations. Here, several basic strategies of server implementation behavior can appear:

- *single thread* – all incoming requests are serialized and processed by a single thread.
- thread per request – for every incoming request a new thread is created. After processing a request, the thread terminates.

- pool of threads – during the start of the server, a pool of threads is created to handle incoming requests. Each thread waits for an incoming request, handles it and loops back to repeat this sequence. This strategy allows for parallel processing, while limiting the server’s use of resources.

- thread per client – a separate thread is created for each client process that connects to a server. This could be useful e.g. when consecutive requests from the same client form a single transaction. A slight modification of this strategy is thread per client connection.

- thread per object – a separate thread is created to handle requests for one server object only (or for a subset of server objects). This could be useful e.g. in real-time processing, where the thread associated with one object has higher priority than the others.

Clearly, it is helpful when the ORB supports at least some of the strategies listed above. It is also important that the ORB is flexible enough to allow other (proprietary) strategies to be implemented.

**Multi-threaded Clients**

Apart from employing multi-threaded servers, there are at least two strong benefits from employing multi-threaded clients:

- non-blocking call – a special thread is created to make a (time consuming) remote operation. The client can continue processing without waiting for the result of the operation. Similar semantics can be achieved using (several) oneway or deferred operations.

- call-back receive – a special thread can be created to handle all incoming call-backs without having to poll for communication events.

**Multi-threaded ORB**

Finally, the issue of employing multi-threading inside the underlying ORB is also important. Here, two questions arise:

- does an ORB employ threads to process its internal actions?

- what is an ORB internal policy for managing TCP/IP connections between clients and servers?

For client or server programmers, a possibility to setup (or change the default) internal ORB behavior is also important.

**Concurrency**

Closely related to the multi-threading is the issue of concurrency control. The shared data structures have to be protected to avoid corruption by
parallel processing. This protection has to apply both at the ORB level and at the application level.

4.2.3. **Management Extensions**

In most cases, the installation of an ORB running on a single computer (or in a relatively small network) is simple to configure and manage. With the growing number of network hosts running ORB clients and servers, however, the management becomes complicated and a need for a sophisticated configuration and management tool arises. As there are no OMG standards for ORB configuration and management, the configuration and management tools (when provided) are proprietary and offer functions specific to a given ORB.

The ORB configuration and management tools (if any) are described in this section.

4.3. **Benchmarks**

4.3.1. **Measurement Verification**

To verify the validity of the benchmark results, it is important to test whether the areas in which the benchmarks are over-simplified (e.g. calling the same object over and over again, calling the object in predictable patterns) do not influence the results. Thus, a simple suite of benchmarks that test the dependence of the invocation times on the invocation strategy used in the benchmark is devised.

**PATTERN**

The benchmark employs the following five invocation strategies (regardless of strategy, the total number of objects and calls used in the benchmark remains constant):

- **As Thru, Rev Thru** – a single object is called 10 times, this is repeated for 1000 objects in their creation order and reverse creation order (this is similar to the behavior of e.g. the *THROUGH.IN* benchmark – see section 4.3.3).

- **As Proxy, Rev Proxy** – 1000 objects are called once in their creation order and reverse creation order, this is repeated 10 times (this is similar to the behavior of e.g. the *PROXY* benchmark – see section 4.3.5.1).

- **Random** – 10000 calls are made to 1000 objects, targets are randomly selected.

The results of the benchmark show the influence of the invocation pattern on the results.
4.3.2. Dispatcher Performance

The nature of the IDL mapping to an implementation language (C++ in my case) and of the IIOP protocol implies dependencies between the structural complexity of the IDL interfaces and the performance of the applications based on these interfaces. The exact nature of this relationship depends mostly on the quality of the ORB dispatcher code (partially a part of the object adapter, partially produced by the IDL compiler).

The IDL. NAMES, IDL. LONG and IDL. DEEP benchmarks are aimed at discovering weaknesses within the dispatcher code, especially with respect to the complexity of dispatching algorithms (e.g. linear searches through a potentially large amount of data).

IDL. NAMES

In the IDL. NAMES benchmark, a single IDL interface within a single IDL module is defined. The interface defines several operations that differ in the length of their name, in a single character at the beginning of the operation name or at the end of the operation name, in the length of the argument name:

```
module test
{
    interface iface
    {
        void x ();
        void x2345...500chars...7890 ();
        void xAAAA...500chars...AAAA ();
        void yAAAA...500chars...AAAA ();
        void AAAAA...500chars...AAAx ();
        void AAAAA...500chars...AAAy ();
    }
};
```

The round trip time (RTT) needed to return from an invocation of the operations is measured.

IDL. LONG

A single IDL interface with a large number of operations is defined:

```
module test
{
    interface ifaceLong
    {
        void x1();
        void x2();
        ...
        void x99();
    }
};
```
The difference in RTT when invoking operations from different positions within the interface is measured.

**IDL.DEEP**

Several interfaces are defined in different depths of an IDL module:

```idl
module test {
    interface iface1 { void x(); }; 
    module mod2 {
        module mod3 {
            module mod4 {
                module mod5 {
                    interface iface5 { void x(); }; 
                    module mod6 {
                        module mod7 {
                            module mod8 {
                                module mod9 {
                                    ...
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

The difference in RTT between invocations of interfaces from different depths in the module is measured.

### 4.3.3. Performance

**IDL.ENCAP**

To compare the times needed to pass data individually, in an array, in an array, in a bounded sequence, and in a structure, the *IDL.ENCAP* benchmark was made. The benchmark compares invocation time of the following four operations:

```idl
typedef long Arr [100];
typedef sequence <long, 100> Seq;
struct Str { long X0; ... long X99; };

interface Encap {
    void NoEncap (in long X0, ... in long X99);
    void ArrEncap (in Arr A);
    void SeqEncap (in Seq S);
    void StrEncap (in Str S);
};
```
THROUGH.IN

To provide information on how fast does an ORB transfer data, a series of benchmarks on passing of a large amount of arguments is included. In these benchmarks, the basic data types (integers, reals, characters, octets, anys) are transferred as separate entities, within arrays, or within sequences. The arrays and sequences (bounded sequences are used) all encapsulate 1kB (1024 bytes) of data, regardless of the type transferred.

```
module test {
    typedef float float_arr [256];
    typedef double double_arr [128];
    ...
    typedef sequence<float,256> float_seq;
    typedef sequence<double,128> double_seq;
    ...
    typedef string<1024> str;
}

interface through {
    long in_float (in float x);
    long in_double (in double x);
    ...
    long in_floatArray (in float_arr x);
    long in_doubleArray (in double_arr x);
    ...
    long in_floatSeq (in float_seq x);
    long in_doubleSeq (in double_seq x);
    ...
    long in_string (in str x);
};
}
```

The results of the benchmark show both the throughput of the ORB and the overhead associated with individual data types being passed. Both the time of delivery of the request and RTT of the request completion are measured by the benchmarks\(^2\) (the time of delivery hints at the raw throughput of the ORB implementation, while the RTT is what interests client application programmers).

The benchmarks are collectively referred to as THROUGH.IN. The measurement results are split into groups covering e.g. the basic types, sequences, arrays.

\(^2\) The delivery time is measured in local configurations only.
4.3.4. *Multi-threading Extensions*

**MT.CLIENT**

The *MT.CLIENT* benchmark tests the multithreading capabilities of the ORB client when multiple threads access different servers. The benchmark uses five servers, each providing a synchronous operation that takes 10 seconds to complete. The client creates five threads, each of the threads invokes the operation five times on one server. The benchmark takes 50 seconds to complete when the client issues the requests from different threads in parallel, and 250 seconds when the requests are serialized.

**MT.CONN**

The *MT.CONN* benchmark tests the multithreading capabilities of the ORB client when multiple threads access the same server. The benchmark is similar to *MT.CLIENT*, but uses only one server providing five objects with a synchronous operation that takes 10 seconds to complete. The client creates five threads, each of the threads invokes the operation five times on one object. The benchmark takes 50 seconds to complete when the client issues the requests from different threads in parallel, and 250 seconds when the requests are serialized.

4.3.5. *Scalability*

In this section, I analyze various aspects of the ORB’s scalability that were not touched by previous benchmarks. Three aspects of scalability are mentioned – speed with respect to number of objects, resource consumption with respect to number of objects, and resource consumption with respect to the number of incoming (asynchronous) requests.

4.3.5.1. *Speed vs. Number of Objects*

An important part of the remote invocation mechanism defined by CORBA is the ability to pass objects by reference. Whenever an ORB receives a new object reference, a proxy object is created and passed to the target application. The application can then invoke operations upon the proxy (these get forwarded to the object represented by the proxy).

Both the time needed to create the proxy object and the time needed to open a connection to the object represented by the proxy vary in different ORBs. The performance of some of the operations performed by the ORB (dispatching, object reference unmarshalling, proxy creation) depends on the number of existing objects and proxies as well.

**PROXY**

The *PROXY* benchmark measures the time it takes to create and return a sequence of objects and to invoke an operation on these objects (the first and all subsequent invocation times differ) This is measured for 1000,
2000 and 3000 objects on the server. The measurement results are split into three groups covering the proxy creation time (PROXY.CREATES) and the difference between the first (PROXY.FIRSTE) and second (PROXY.NEXT) call to a newly created proxy.

**PROXY.LOT**

The *PROXY.LOT* benchmark was designed to test the degradation in request delivery speed when handling a large number of objects. A client and two servers are involved in the test – the client and the first server increase the number of their objects by a thousand each test run, while the second server keeps its number of objects constantly low. A difference in invocation speed is measured for two operations:

```c
interface helo { ... }

typedef sequence<helo> helo_seq;

interface factory {
    helo_seq get (in long x);
    long nop ();
    long op5 (in helo h1, ... in helo h5);
};
```

**4.3.5.2. Resources vs. Number of Objects**

The consumption of memory per object on both the server and the client is measured using a slightly modified code of the *PROXY* benchmark.

**4.3.5.3. Resources vs. Queued Requests**

**ONeway**

The *ONeway* benchmark was made to test the number of oneway calls that can be issued until the client blocks. The benchmark client keeps issuing oneway void calls that take a long time to complete at the server side and measures the time it takes to issue the call. Long delays indicate the client being blocked, these are registered by the benchmark.

**4.3.6. Robustness**

This section is dedicated to the issue of ORB robustness. The subsections discuss the support for building reliable servers using the ORB, measure some of the limits of the ORB implementation (size of requests, number of objects). I also list some of the bugs I have found during testing.

**4.3.6.1. Reliable Servers**

The support for building reliable servers using the ORB is discussed in this section.
4.3.6.2. Limits

**SIZE**

This benchmark tries to send data packets of increasing size in a single request to find out if there are any limits imposed on the request size and to test whether the server can cope with large requests.

**Number of Objects**

The maximum number of objects the server and the client are able to cope with is tested using a slightly modified code of the PROXY benchmark.

4.3.6.3. Bugs

Some of the bugs discovered during the testing of the ORB are listed in this section.
5. Conclusion

The goal of the thesis as described in section 1.1 has been reached. The evaluation criteria have been proposed and used to compare set of ORBs (omniORB 2.7.1, ORBacus 3.1.2, Orbix 3.0; see Appendix). They are valuable and give good picture of evaluated ORB. Such evaluated ORBs can be easily compared by user to select the proper implementation.

I have focused on evaluation of single CORBA implementation more than on the comparison moment. This fact has been justified in the thesis – the user that needs comparison should determine what features are important for him and compare them with relevant weights against each other.

Other important moment in the thesis is selecting the strategy to benchmark only on a single platform. This has been justified in sections 3.1.2 and 3.1.3.

In spite of these drawbacks, the thesis still provides important results. As far as I know, the report [20], in which many of the concepts described here were used, is the very first serious attempt at comparing multiple ORB implementations, and has received significant public response.

5.1. Current Status and Future Work

The criteria presented in this thesis were used during the CORBA Comparison Project (participants are The CORBA and Distributed Systems Research Group, Charles University and MLC Systeme, GmbH, Germany as the industrial partner) and after being published on the Internet, the project report [20] drew a lot of attention. It is worth noting that the responses were only positive.

A general test suite can be developed to make the overall evaluation easier. To make such suite, however, many of problems with benchmarking outlined here need to be solved. As the top priority for the nearest future, I would attempt to make the benchmark results comparable. This can be done by factoring more complex measurements
into simpler ones, or by defining a formula that would express the results in comparable units.

5.2. Related Work

An early attempt at comparing CORBA implementations can be found in [24], dated 1996. The attempt forms a simple set of criteria, which are, in the author’s point of view, essential for constructing large-scale enterprise systems. Another serious attempt is described in [25], where several ORB implementations are mutually compared. In the following years, several other attempts appeared, mostly from the ORB vendors themselves (simple tests showing the ORB performance compared to a couple of other ORBs). The most serious step ahead was publishing the CORBA Comparison Project report [20] on the Internet. This report is significantly more thorough than any of the comparison attempts existing before, and the work outlined in this report is also a core of my thesis. Following this, several works based on similar tests were published [26].

The work of Douglas Schmidt from Washington University, St. Louis, U.S.A., deserves a special attention. It includes ORB implementation (which serves as referential model) and series of articles on CORBA and distributed computing. Some of the articles focus on individual ORB components (e.g. multi-threading, object adapter) or on overall performance (e.g. [8], [9]).
6. References


[17] *University of Helsinki Response to the Benchmark RFI*, University of Helsinki, OMG Document bench/98-10-03, October 1998


Appendix

The appendix to the thesis contains several ORB implementations evaluation. Simple general comparison is included at the end.